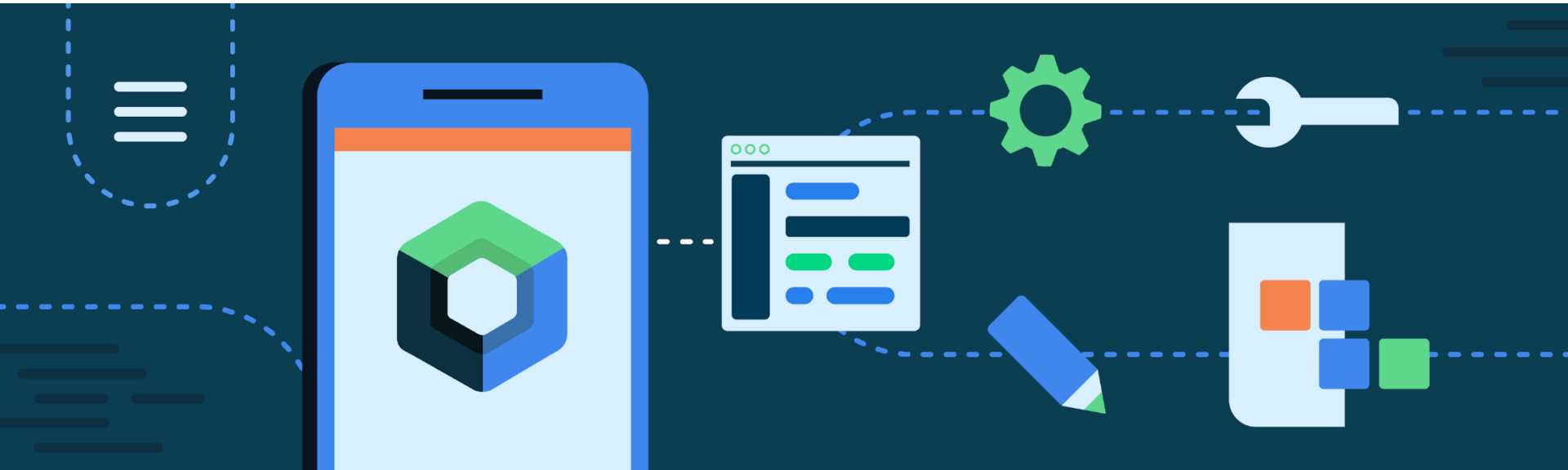


# CMPS 312



## Declarative UI using Jetpack Compose

Dr. Abdelkarim Erradi  
CSE@QU

# Outline

1. Jetpack Compose Key Concepts
2. UI Components
3. Modifiers
4. Layouts
5. State

# Jetpack Compose Key Concepts



<https://developer.android.com/jetpack/compose/mental-model>

# Declarative UI is a major trend

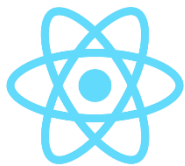
- Describe WHAT to see NOT HOW



Flutter: Google's UI toolkit for building natively compiled applications for mobile, web and desktop from a single codebase



SwiftUI: Apple's new declarative framework for creating apps that run on iOS



React: A JavaScript library for building user interfaces



Jetpack Compose: a **modern toolkit** for building native Android UI ([released July 2021](#))

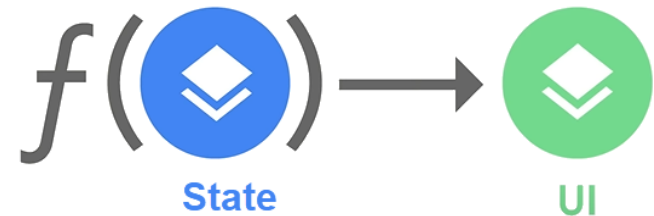
# Jetpack Compose

- Jetpack Compose is a **modern UI toolkit** for Android
  - It simplifies UI development with less code and intuitive Kotlin APIs that follow **best practices**
- **A declarative component-based programming model**
  - UI is built using composable functions
    - Each function define a piece the app's UI programmatically by **describing WHAT to see** (layout/ look and feel) **NOT HOW**
  - As state changes the UI automatically updates (Reactive UI) (without imperatively mutating UI views)
  - Inspired by/similar to other declarative UI frameworks such as React and Flutter



# How to define a piece of UI?

- UI is **composed** of small reusable **components**
- UI Component = Composable **function**:
  - Just a function annotated with **@Composable**
  - Takes some inputs and emits a piece of UI
  - Describes the desired screen state (**WHAT to see**)
    - Compiler takes care of the HOW and constructs UI widgets
  - Converts the input data into UI



- **UI = f(state) : UI is a visual representation of state**  
(e.g., display a tweet and associated comments)



- **State changes trigger automatic update of the UI**

# UI as a function

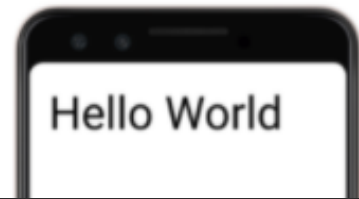
String



```
fun Greeting(name: String) =  
    println("Hello, $name")
```



stdout



Mark as a composable

Data

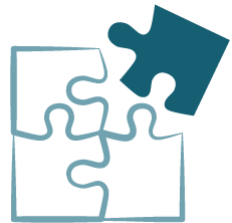


```
@Composable  
fun Greeting(name: String) =  
    Text("Hello, $name")
```



UI

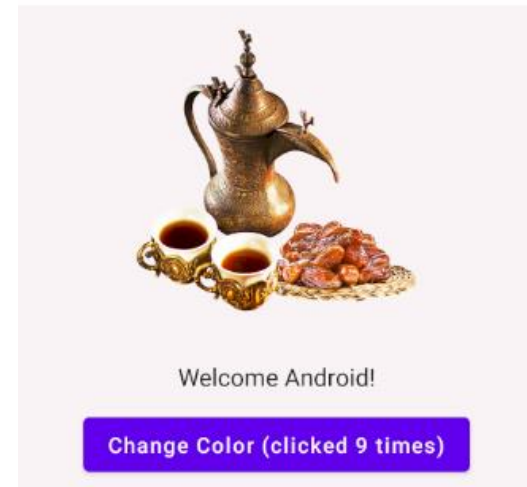
**Greeting** function uses the input data to render a Text widget on the screen



# UI = Composition of UI functions

Your name

- The top-level composable function describes the UI by calling other composables and passing them the appropriate data



@Composable

```
fun WelcomeScreen() {  
    var userName by remember { mutableStateOf( value: "Android") }  
    Column { this: ColumnScope  
        | NameEditor(name = userName, nameChange = { newName -> userName = newName })  
        | Welcome(userName)  
    }  
}
```

@Composable

```
fun NameEditor(name: String, nameChange: (String) -> Unit) {...}
```

@Composable

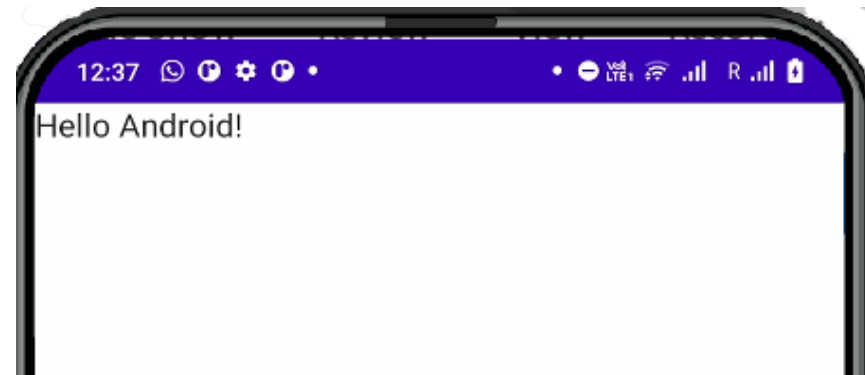
```
fun Welcome(name: String) {...}
```



# App Entry Point

- When the app launches it creates and starts the **Main Activity** (specified in *AndroidManifest.xml*)
- The **Activity** acts as a container to load the UI main screen using **setContent** in the **onCreate** method

```
class MainActivity : ComponentActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            Greeting("Android")  
        }  
    }  
}  
  
@Composable  
fun Greeting(name: String) {  
    Text(text = "Hello $name!")  
}
```

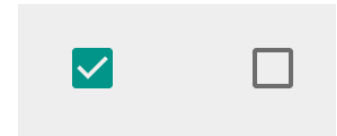


# UI Components

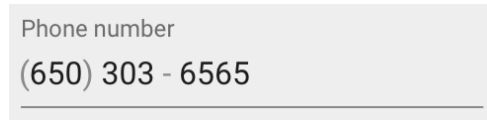
Button



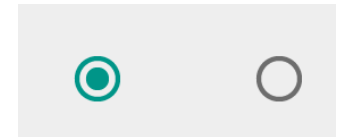
CheckBox



TextField



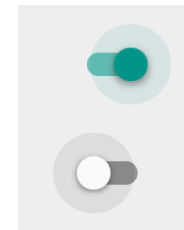
RadioButton



Slider



Switch



# Text box

- **Text()** displays a simple text

```
Text(  
    text = "Jetpack Compose",  
    style = MaterialTheme.typography.h4  
)
```

Jetpack Compose

```
Text(  
    text = "سور القرآن الكريم",  
    textAlign = TextAlign.Center,  
    modifier = Modifier.fillMaxWidth(),  
    style = TextStyle(  
        fontWeight = FontWeight.Bold,  
        fontSize = 24.sp,  
        color = Color.Blue,  
        textDirection = TextDirection.Rtl  
    )  
)
```

سور القرآن الكريم

# TextField

- **TextField()** collects input from a user. For more styling options, use **OutlinedTextField()**

@Composable

```
fun NameEditor(name: String, onNameChange: (String) -> Unit) {  
    OutlinedTextField(  
        value = name,  
        onChange = onNameChange,  
        label = { Text("Your name") }  
    )  
}
```



# Button

```
Button(onClick = {}) {  
    Text("Button")  
}
```

```
OutlinedButton(onClick = {}) {  
    Text("OutlinedButton")  
}
```

```
TextButton(onClick = {}) {  
    Text("TextButton")  
}
```

```
// Search icons @ https://fonts.google.com/icons  
IconButton(onClick = { }) {  
    Icon(  
        Icons.Outlined.Search,  
        contentDescription = "Search",  
    )  
}
```

```
IconButton(onClick = { }) {  
    Icon(painterResource(id = R.drawable.ic_quran), "Quran")  
}
```

Button

OutlinedButton

TextButton



# Image

- Displays an image from the res/drawable folder

```
Image(painter =  
    painterResource(R.drawable.img_compose_logo),  
    contentDescription = "Jetpack compose logo",  
    modifier = Modifier.height(300.dp))
```



# Other Basic UI Components

- **RadioButton()** allows selecting from multiple choices
- **CheckBox()**

# Layouts



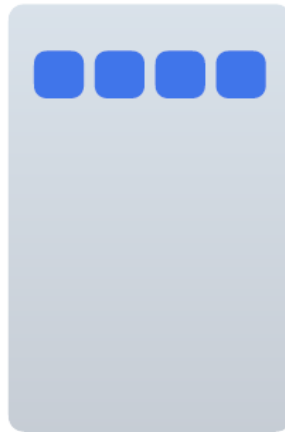


# Layouts

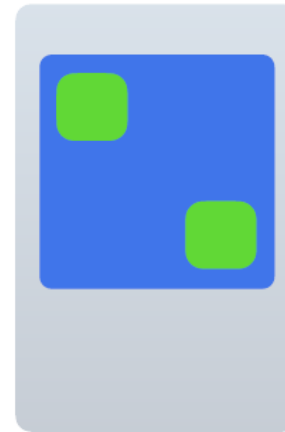
- Use a Layout to **position** UI elements on the screen
- **Row** - position elements horizontally
- **Column** - position elements vertically
- **Box** - position elements in the corners of the screen or stack them on top of each other
- Use [Constraint Layout](#) (self-study) for complex layouts



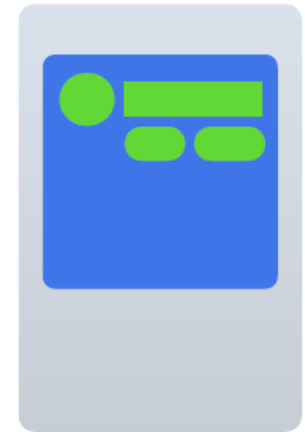
Column



Row



Box



Constraint  
Layout

# Row & Column Example

- Group multiple basic layouts to create a more complex screen
- Use vertical or horizontal **Arrangement** to change the position of elements inside the Row or Column

```
@Composable
fun ArtistCard(artist: Artist) {
    Row(verticalAlignment = Alignment.CenterVertically) {
        Image(/*...*/)
        Column {
            Text(artist.name)
            Text(artist.lastSeenOnline)
        }
    }
}
```



**Alfred Sisley**

3 minutes ago

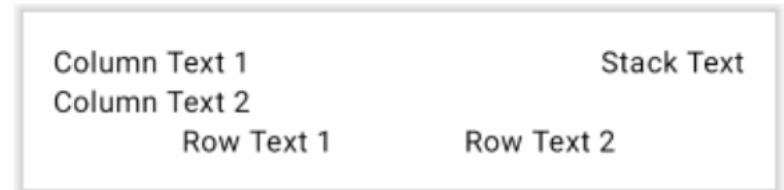
# Box Example

```
@Composable
fun ArtistAvatar(artist: Artist) {
    Box {
        Image(/*...*/)
        Icon(/*...*/)
    }
}
```



# Box Example (1 of 4)

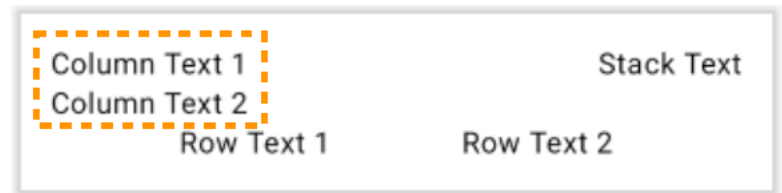
```
Box(modifier = Modifier.fillMaxWidth()) {  
    Column(  
        modifier = Modifier  
            .padding(16.dp)  
            .fillMaxWidth()  
    ) {  
        Text("Column Text 1")  
        Text("Column Text 2")  
  
        Row(  
            modifier = Modifier.fillMaxWidth(),  
            horizontalArrangement = Arrangement.SpaceEvenly  
        ) {  
            Text(text = "Row Text 1")  
            Text(text = "Row Text 2")  
        }  
    }  
    Text(  
        "Stack Text",  
        modifier = Modifier  
            .align(Alignment.TopEnd)  
            .padding(end = 16.dp, top = 16.dp)  
    )  
}
```



## Box Example (2 of 4)

```
Box(modifier = Modifier.fillMaxWidth()) {
    Column(
        modifier = Modifier
            .padding(16.dp)
            .fillMaxWidth()
    ) {
        Text("Column Text 1")
        Text("Column Text 2")

        Row(
            modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.SpaceEvenly
        ) {
            Text(text = "Row Text 1")
            Text(text = "Row Text 2")
        }
    }
}
Text(
    "Stack Text",
    modifier = Modifier
        .align(Alignment.TopEnd)
        .padding(end = 16.dp, top = 16.dp)
)
}
```



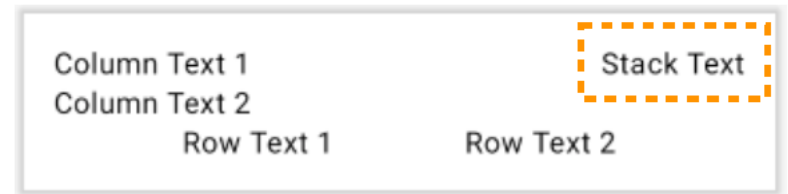
# Box Example (3 of 4)

```
Box(modifier = Modifier.fillMaxWidth()) {  
    Column(  
        modifier = Modifier  
            .padding(16.dp)  
            .fillMaxWidth()  
    ) {  
        Text("Column Text 1")  
        Text("Column Text 2")  
  
        Row(  
            modifier = Modifier.fillMaxWidth(),  
            horizontalArrangement = Arrangement.SpaceEvenly  
        ) {  
            Text(text = "Row Text 1")  
            Text(text = "Row Text 2")  
        }  
    }  
    Text(  
        "Stack Text",  
        modifier = Modifier  
            .align(Alignment.TopEnd)  
            .padding(end = 16.dp, top = 16.dp)  
    )  
}
```



# Box Example (4 of 4)

```
Box(modifier = Modifier.fillMaxWidth()) {  
    Column(  
        modifier = Modifier  
            .padding(16.dp)  
            .fillMaxWidth()  
    ) {  
        Text("Column Text 1")  
        Text("Column Text 2")  
  
        Row(  
            modifier = Modifier.fillMaxWidth(),  
            horizontalArrangement = Arrangement.SpaceEvenly  
        ) {  
            Text(text = "Row Text 1")  
            Text(text = "Row Text 2")  
        }  
    }  
    Text(  
        "Stack Text",  
        modifier = Modifier  
            .align(Alignment.TopEnd)  
            .padding(end = 16.dp, top = 16.dp)  
    )  
}
```



# Surface & Card

- A **Surface** can hold only one child with an option to add a border and elevation
  - Add a layout inside Surface to position multiple elements
- A **Card** is a just a Surface with default parameters



# Responsive Layout

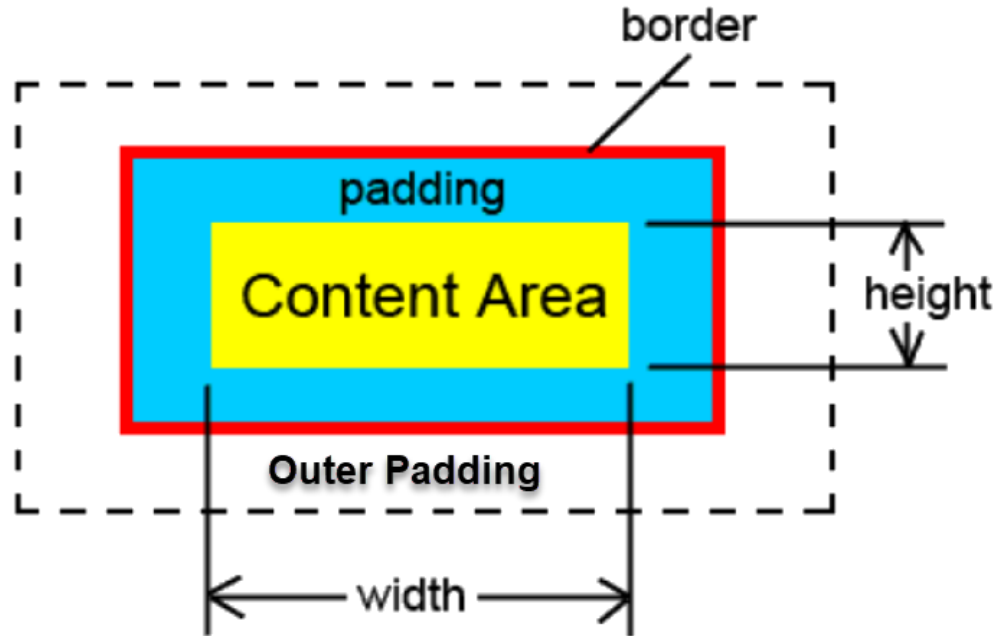
- Use the **weight** modifier in Row and Column layouts:
  - Use **weights** to change the **proportion** of the screen child elements will use
  - Distribute space among items in a container while accommodating different screen sizes
- Modifier.*fillMaxWidth*() fill available width
- Modifier.*fillMaxHeight*() fill available height
- Modifier.*fillMaxSize*() fill available width and height
- Use [Constraint Layout](#) (self-study) for more control for complex scenarios

# Modifiers

# Modifiers

- Modifiers are used to configure and customize the style (i.e., look) or behavior of UI components
  - **Style** UI element such as colors, borders, paddings, **layout parameters** to control spacing and positioning
  - **Add behavior** to UI elements such as making the element clickable or scrollable
- Several modifiers can be **chained**
  - Each modifier **modifies** the composable and **prepares** it for the next modifier in the chain
  - The **order of modifiers** in the chain matters

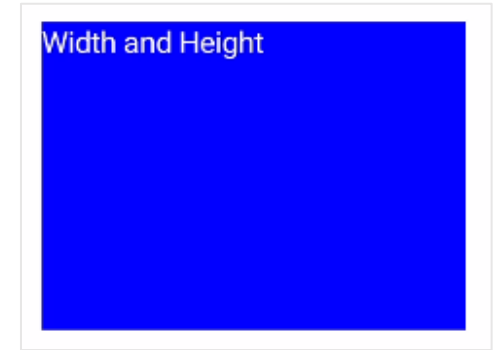
# Size and Spacing



- Composable size and spacing properties can be styled:
  - **Outer padding** (i.e., margin) - the space that separates composables
  - **Border** - the line around each edge of the composable
  - **Padding** - the space between the border and the content

# Size and Spacing - Example

```
Text(  
    text = "Width and Height",  
    color = Color.White,  
    modifier = Modifier  
        .padding(10.dp) // Outer padding (margin)  
        .background(Color.Blue)  
        .width(200.dp)  
        .height(150.dp)  
    // .size(width = 250.dp, height = 100.dp) // Alternative way  
)
```



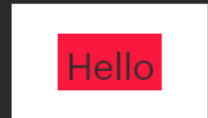
```
Text(  
    text = "Padding and margin!",  
    modifier =  
        Modifier.padding(16.dp) // Outer padding (margin)  
            .background(color = Color.Yellow) // background color  
            .border(  
                width = 2.dp,  
                color = Color.Gray  
            ) // Add a border  
            .padding(8.dp) // Inner padding  
)
```



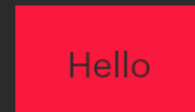
# Modifiers Chain

- Modifiers can be chained and the order matters!
  - Applied in a sequential way and the order impacts the behavior

```
Text(  
  text = "Hello",  
  modifier = Modifier.padding(16.dp)  
    .background(color = Color.Red)  
)
```



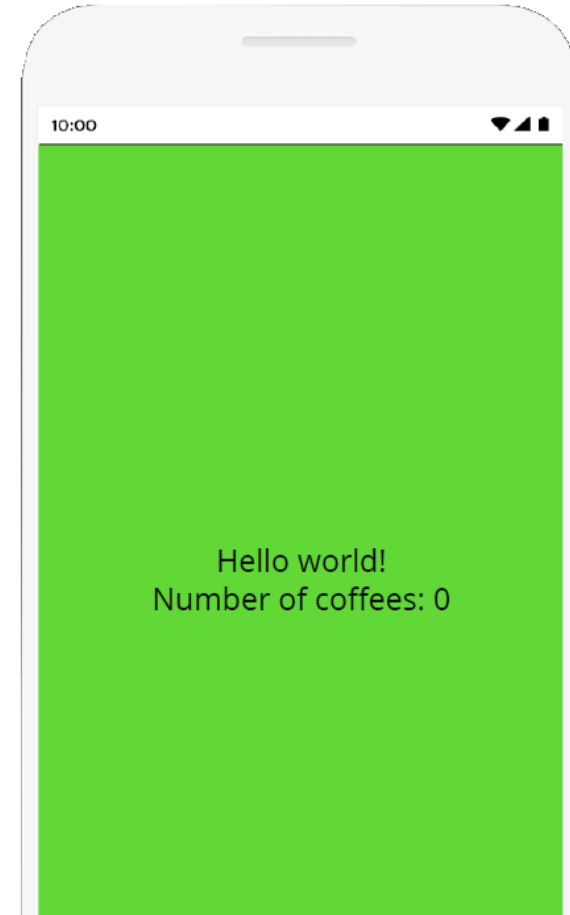
```
Text(  
  text = "Hello",  
  modifier = Modifier.background(color = Color.Red)  
    .padding(16.dp)  
)
```



# Another Modifier Example

```
@Composable
fun Greeting(name: String) {
    Column(
        modifier = Modifier.fillMaxSize()
                           .background(Color.Green)
                           .padding(16.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ){
        Text(text = "Hello $name!")
        Text(text = "Number of coffees: 0")
    }
}
```

Kt



# Surah Card



110. النصر - An-Nasr

Aya count: 3

@Composable

```
fun SurahCard(surah: Surah) {  
    Card (elevation = 10.dp,  
        backgroundColor = if (surah.type == "Medinan") lightGreen else lightYellow,  
        modifier = Modifier  
            .fillMaxWidth()  
            .padding(horizontal = 5.dp)  
            .border(width = 2.dp, color = Color.LightGray, shape = RoundedCornerShape(8.dp))  
    ) {  
        Row (verticalAlignment = Alignment.CenterVertically,  
            horizontalArrangement = Arrangement.spacedBy(4.dp),  
            modifier = Modifier.padding(5.dp)  
        ) {  
            val imgResourceId = if (surah.type == "Medinan") R.drawable.ic_madina  
                                else R.drawable.ic_mecca  
            Image(painter = painterResource(id = imgResourceId),  
                contentDescription = "Surah Type",  
                Modifier.height(50.dp)  
            )  
            Column(verticalArrangement = Arrangement.spacedBy(2.dp)) {  
                Text(text = "${surah.id}. ${surah.name} - ${surah.englishName}")  
                Text(text = "Aya count: ${surah.ayaCount}")  
            }  
        }  
    }  
}
```



# Modifier.*clickable*

```
Text(  
    text = "+",  
    modifier = Modifier  
        .border(2.dp, Color.Gray)  
        .padding(10.dp)  
        .clickable {  
            count += 1  
        },  
    style = MaterialTheme.typography.h5  
)
```



# State




<https://developer.android.com/jetpack/compose/state>

# State

- State = any value that can change overtime
- State variable must be declared as

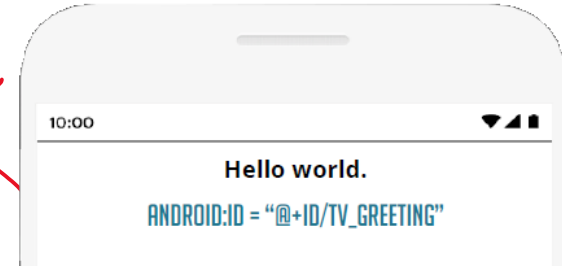
```
var stateVar by remember { mutableStateOf(default) }
```

- **Remember** is used to **store** values of state variable in the composition tree (to preserve the values during the recomposition)
    - the stored value is returned during recomposition
  - State variable **are observed** by the Jetpack compose runtime
    - Any value changed in the state will trigger the recomposition of any composable functions that **read** the state variable
-  ○ Every place a state variable is displayed is guaranteed to be auto-updated

# Imperative UI vs. Declarative UI

- Imperative UI – call a setter on the view to change its internal state

```
TextView greetings = (TextView) findViewById(R.id.tv_greeting)  
greetings.text = "Hello world."
```



- UI in Compose is immutable
  - In compose you cannot access/update UI elements directly (as done in the imperative approach)
  - The only way to update the UI is by updating the state variable(s) used by the UI elements – this triggers automatic UI update
    - E.g., displayed ***greeting text*** can only be changed by updating the ***name*** state variable

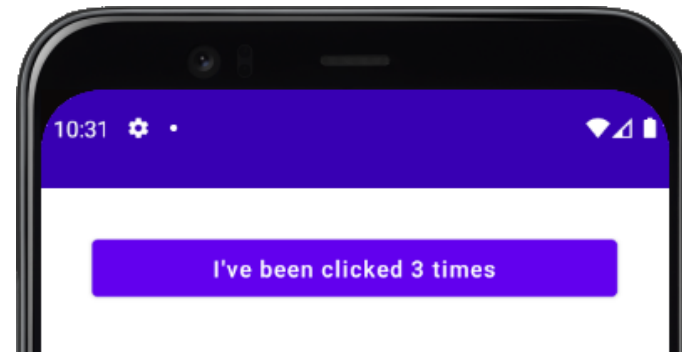
```
@Composable  
fun WelcomeScreen() {  
    var name by remember { mutableStateOf("Android") }  
    Greeting(name)  
}
```

```
@Composable  
fun Greeting(name: String) {  
    Text(text = "Hello $name!")  
}
```

# Recomposition

- When the user interacts with the UI, the UI raises events such as onClick
  - Those events should notify the app logic, which can then change the app's state
  - When the state changes it causes the composable functions to be automatically called again with the new data => this causes the UI elements to be redrawn
  - This process is called **recomposition**
- The Compose framework can intelligently recompose only the components that changed

# Recomposition Example



- Every time the button is clicked, the UI raises **onClick** event to notify the app logic, which increments **clicksCount** state variable
- This causes a **recomposition** to take place, i.e., the **ClickCounter** function is automatically called again to redraw the Button

@Composable

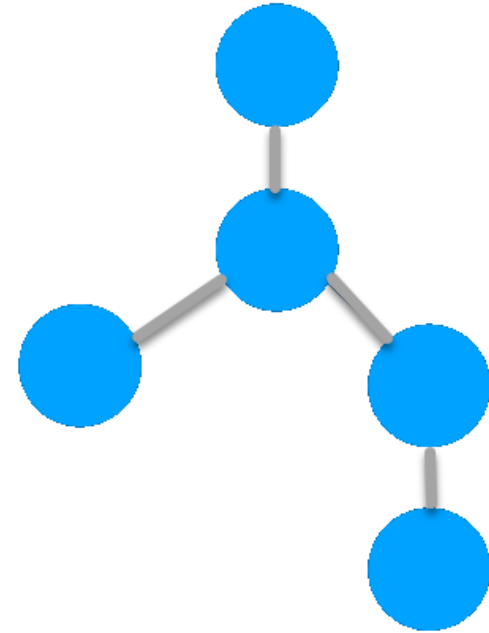
```
fun MainScreen() {  
    var clicksCount by remember { mutableStateOf(0) }  
    ClickCounter(clicks = clicksCount, onClick = { clicksCount += 1 })  
}
```

@Composable

```
fun ClickCounter(clicks: Int, onClick: () -> Unit) {  
    Button(onClick = onClick) {  
        Text("I've been clicked $clicks times")  
    }  
}
```

# How recomposition works

1. Creates an abstract representation of the UI and renders it
2. When a change occurs, it creates a new representation
3. Computes the differences between the two representations
4. Renders the differences [if any]



For more details about [Jetpack Compose Runtime](#), watch this [video](#)

# Stateful versus stateless

- A stateful composable uses **remember** to store an object in the composition tree
  - However, stateful composable tend to be less reusable and harder to test
- A stateless composable that doesn't hold any state
  - The caller controls and manages the state
  - An easy way to achieve stateless is by using **state hoisting**



# State Hoisting

- To make a composable stateless, **extract** its state and **move it to the caller** of the composable
- Then **pass the state** to the composable as an immutable parameter, along with a callback function that the UI can call to update that state in response to events (e.g., `onValueChange`, `onExpand` and `onCollapse`):
  - **`name: String`** - the current value to display
  - **`onNameChange: (String) -> Unit`** - a callback that requests the value to change
- Hoisted state can be shared with multiple composables

# State Hoisting - Example

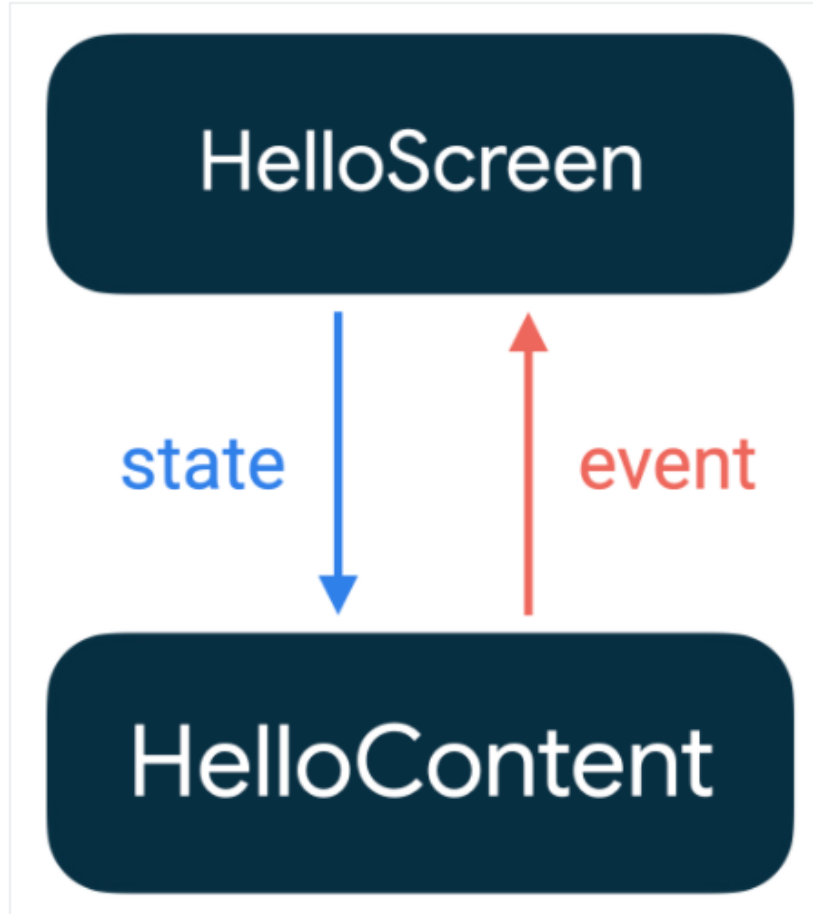
```
@Composable
fun HelloScreen() {
    var name by remember { mutableStateOf("") }

    HelloContent(name = name, onNameChange = { name = it })
}

@Composable
fun HelloContent(name: String, onNameChange: (String) -> Unit) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(
            text = "Hello, $name",
            modifier = Modifier.padding(bottom = 8.dp),
            style = MaterialTheme.typography.h5
        )
        OutlinedTextField(
            value = name,
            onValueChange = onNameChange,
            label = { Text("Name") }
        )
    }
}
```

# Unidirectional Data Flow

= a design where **state flows down** and **events flow up**



**State flows down via function parameters**

(i.e., name)



**(State change)**

**Events flow up via callback functions**

(i.e., onNameChange)

By hoisting the state out of HelloContent, it can be **reused** in different situations, and it is easier to test

# Summary

- Declarative UI is the trend for UI development
- UI is composed of small reusable components
- UI Component = Composable function
  - just a function annotated with `@Composable`
- Layout are used to position UI elements
- UI in Compose is immutable
  - It only accepts state & exposes events
  - Unidirectional data flow pattern:
    - State flows down via parameters
    - Events flow up via callbacks
- .. mastering Compose will take some time and effort   ...

# Resources

- Jetpack compose tutorial

<https://developer.android.com/jetpack/compose/tutorial>

- Jetpack compose Code Labs

<https://developer.android.com/courses/pathways/compose>

- Jetpack Compose Playground - UI component examples

<https://foso.github.io/Jetpack-Compose-Playground/>

<https://github.com/Foso/Jetpack-Compose-Playground>

<https://github.com/Gurupreet/ComposeCookBook>

- Compose Samples

<https://github.com/android/compose-samples>